University of Edinburgh

Department of Computer Science

The Structure of the EMAS 2900 Kernel

by

D.J. Rees

CSR-91-81

August, 1981

James Clerk Maxwell Building,
The King's Buildings,
Mayfield Road,
Edinburgh,
EH9 3JZ.

The Structure of the EMAS 2900 Kernel

D. J. Rees

Department of Computer Science
Edinburgh University

The role of the kernel of the Operating System EMAS 2900 and the implementation of its functions is described in some detail. The significance of local scheduling policies and their implications on the design of the kernel are discussed with particular reference to paging management and scheduling control. It is shown that the concept of local and global control of resources can lead to a considerable simplification in the structure of an operating system kernel. The resulting EMAS 2900 provides time-sharing services very effectively and efficiently to a large computing community.

Introduction

EMAS 2900 is a multi-access time-sharing system for the ICL 2900 series of computers which was developed by a small group of staff from the Department of Computer Science and the Edinburgh Regional Computing Centre in Edinburgh University. The development was in essence a reimplementation of the EMAS system which ran on the ICL System 4-75 computer using the same underlying philosophy for the most part but taking into account the experience of several years use and improvement it had undergone and new insights which had resulted. As with any large system, the EMAS implementation had become somewhat untidy and more difficult to maintain as time went by and as the original team left the scene. A significant goal was therefore to achieve a simplification and "cleaning-up" of the system in order to facilitate future in-service developments and improvements and to postpone the inevitable point in time when complexity escalation and the consequent poorly-understood interactions make further changes difficult to contemplate. There was also a strong desire to investigate and exploit the architecture of the 2900 series in relation to multi-access systems. An overall view of this project has been described by Stephens et al. [ST80]. The purpose of the present paper is to describe the kernel of the system in some detail, in particular the organisation of the virtual memory control and scheduling.

The objectives and structure of EMAS were described by Whitfield et al. [WH73] and much of that has been carried forward to EMAS 2900. The objectives remain very similar, namely, to provide a large-scale interactive system which both gives good facilities and response to users and makes efficient use of the hardware resources. Technological developments since the days of System-4 have changed some aspects. The trend, for instance, towards machines with very much larger main stores and away from drum storage as a paging medium has affected the approach to scheduling. From the standpoint of the kernel, the most significant features are the provision of a virtual machine for each of a large number of users, the controlled sharing of information between those virtual machines and the requirement for response and efficiency. A virtual machine in EMAS 2900 consists of a virtual address space and a virtual processing unit which provides access to the non-privileged instruction set and to a wide variety of system services including input-output services and a comprehensive file storage system. The term "process" is used here to signify the operation of such a virtual machine. What the user sees at his console is at a much higher level than this since he is insulated from the raw virtual machine by a "sub-system" which provides a command interpreter and a convenient interface to the system services.

One of the most important design concepts in both EMAS systems has been that of process-local page replacement policies. Overall performance has vindicated this choice in comparison with systems using global policies and theoretical studies also bear out the wisdom of this choice [DE78]. The full recognition of the significance of this policy motivated what is perhaps the main difference in structure between the two EMAS systems. Whereas in EMAS the implementation of the local policies was intermingled with the rest of the resident kernel, in the EMAS 2900 system a very clear separation has been made between local policy controllers and controllers of global functions. In principle, each process contains an incarnation of a local controller whose

function is to control those and only those resources which have been allocated to that process from the global scheduling controller. This notion of completely separated local controllers has also been utilised with very great benefit in the design of the communications sub-system, described by Laing and Shelness in [LA81].

The implementation of this policy of separation was greatly facilitated by the organisation of 2900 virtual address spaces. Each virtual address space of 2**32 bytes is divided into two halves, a "local" half unique to each process and a "public" half which is shared by all processes. The local half contains all the programs and data in use by the user of that particular process together with an incarnation of the local controller, the director and the sub-system. The director is the innermost layer of software of a process and incorporates many of the local system services such as the file system services. Its functions in EMAS 2900 remain the same as in EMAS which was described in [RE75]. The sub-system implements the next layer of the hierarchy which includes the basic command interpreter, editors, compilers and loaders etc. As much as possible of this material is shared between processes using the standard sharing mechanisms of the system. This includes the director and sub-system code together with all the compilers and editors etc. that the user may happen to be using. The local controller code is also shared but it was found convenient to compile this as a module of the kernel as will be clarified later. The public half of the virtual address space contains the kernel of the system i.e. the global controller, the message passing dispatcher, device handlers etc. The arrangement of processes and controllers is shown in figure 1.

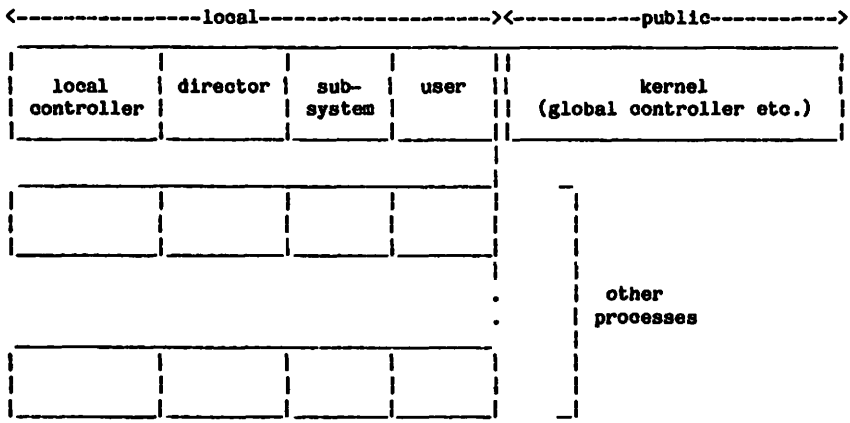

figure 1

The kernel thus appears in every process address space and indeed always runs in virtual mode unlike most earlier hardware designs such as the 4-75 where it ran in real address mode. Switching between the current local space and the kernel therefore does not involve switching virtual machines. Peripheral interrupts, for example, can be directed to an address in the kernel in the same virtual space whilst interrupts such as page faults and local process time-outs can be taken directly by the local controller. There they can immediately be dealt with according to the resources that have been allocated to that process.

The resources in question are primarily pages of physical storage and CPU-time but there are also various other internally defined resources such as "active memory" sections (described below) which also have to be controlled.

The basic page size of the 2900 series architecture is 1K bytes. In the light of the implementors' experience on EMAS we decided that this would probably be too small for best efficiency. To overcome this problem EMAS 2900 groups these basic pages together to form larger units. We had originally hoped to be able to experiment with different unit sizes so as to choose the most efficient but this proved to be infeasible mainly due to the difficulties of varying physical block formats on disc and magnetic tape. The unit size we fixed on was the same 4K bytes that we had used on EMAS and this gave a useful continuity in addition to being what we felt was a sensible choice. "Pages" hereinafter therefor refer to these 4K byte unit multiples of basic pages.

The whole of the operating system is written in IMP, the language used in Edinburgh University for most systems implementation work. This was described by Stephens in [ST74]. The architecture of the 2900 series was designed very much with the use of this kind of high level language in mind [BU78]. In particular, the hardware defines a stack segment which can be used as the stack for IMP storage allocation and procedure calling protocols. A potential problem for operating system kernels is the size of its internal arrays when the number of users and processes is likely to vary considerably either over time or over the various machines in a range such the 2900 series. This has been overcome in the EMAS 2900 kernel by making use of the fact that it runs in virtual space itself although permanently resident. A dynamic scheme is used. Each array that may need to be extended is mapped into a separate segment. Initially, a suitable minimum size is chosen and thereafter physical store pages claimed from the free page-frame list in the ordinary way can be added onto the end of the segment and locked down as and when required. An extra entry is then appended to the appropriate page table.

In order to coordinate the sharing of information, physical movement of pages is initiated and controlled globally but this is in response to requests from local controllers. Each local controller is unaware of any sharing of pages which is taking place between processes. It makes requests purely for certain pages to be made available to it in main store. The global controller then fulfills the request as best it can. For instance, if the page is already in store being used by another process or is still in store from a previous usage but that physical page-frame has not yet been re-allocated, the global controller can simply tell the requesting local controller where the required page is and allow it to continue without having to initiate a page transfer from backing store. This is all transparent to the local controller. Similarly, when a local controller decides that the process it controls no longer needs a particular page, it just tells the global controller and leaves the global controller to get on with removing it while it itself continues. If the global controller knows that another process is still using the page it will not need to page it out. This method of operation in which the local controller only has to be concerned with its own individual process makes implementation of the local controller very much easier and the result much more reliable. The details of its data structures and implementation are described later. The details of the global controller are described first. Its organisation reflects the two functions of paging control and scheduling of resource

allocation. The former can be regarded as including disc and drum handlers but these will not be discussed here as they are relatively straightforward and do not introduce any great originality.

Paging Control

A fundamental feature of EMAS is the way in which virtual memory is used. Conceptually, files are mapped into virtual memory space such that when a particular virtual address is accessed the corresponding item in the file is referenced. The action of creating the mapping between a file and an area of virtual memory is termed "connection". This is purely a logical operation and no file data is transferred around the system at this time. Physical movement of file data is only initiated when a page-fault occurs. This is the well-known "one-level store" concept implemented in many systems. No direct physical movements of file data are ever requested by a user. It is furthermore the case in EMAS that a user does usually not even know which disc his files are stored on. The allocation of disc space is handled entirely behind the scenes by the file system. A file resides on disc storage as a set of disc "extents" (termed "sections" here) as a matter of convenience for the file manager. The connection mapping therefore consists of a table containing the locations on disc of each of the sections of the files which are currently connected. This table forms part of the local controller and will be described in detail later. The significance to paging control lies in this division into sections.

The file is the unit of sharability to EMAS users and all files are potentially sharable, simultaneously, between any number of processes. Since files may potentially be very large, it is more convenient to use the section as the basic unit which the global controller handles rather than the file. When a local controller requires a file page for its process, it must first ensure that the section within which that page lies is "active". The action of "activating" a section takes the form of a request from the local controller to the global controller specifying the disc start address of the pages of the section together with the length of the section and a "new" page mask. The global controller then sets aside an appropriate data structure for this section (an "active memory table entry") and allocates a logical section number, known as an "active memory table index" (amtx), which is used thereafter to identify the section. Again, no physical movement of file data takes place at this time. Individual page requests specify the particular amtx and a page-within-section number. The new page mask allows the local controller to specify that certain pages in the section have never had data written into them. This allows the global controller to avoid making a physical transfer from backing store when such pages are requested. A free store page-frame is simply allocated and cleared to zero. The mask of those pages remaining new is returned to the local controller when the section is deactivated. In principle this means that if the file system were to store this updated mask the disc sites would never need to be cleared to zero for privacy reasons. In practice, the file system clears any non-used page sites to zero on disc when the file is disconnected as this proved to be more convenient. Figure 2 shows the data structure for such an activated section.

The collection of AMT entries for all the active sections in effect forms a dictionary through which sharing can be controlled. It is organised as a hash table using linked lists of entries and with the disc address as the key. Logically speaking, the disc address could be

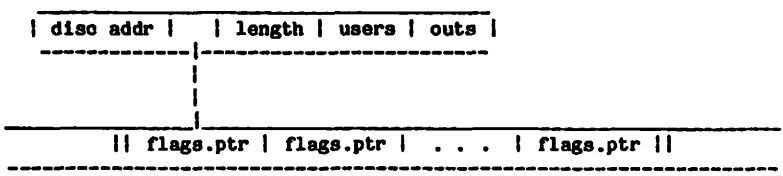

figure 2

used instead of the amtx value to identify the active section but search time is saved by using the latter. Sections shared between processes appear having a "users" count greater than one. The "outs" field contains a count of the number of page-out transfers of pages belonging to this section currently in progress. When the "users" and "outs" counts both drop to zero, the section can be removed from the AMT dictionary. The pointer field shown indicates a collection of entries, one per page in the section, which record the current status of each page. This area is allocated within a single global array and since sections may have different lengths, a dynamic allocation scheme is used to find space. A simple free list is adequate for the fixed length header records.

The status information for each page consists of two flag bits and a pointer field. The first flag bit is the "new" bit distributed from the activation mask and the second indicates whether there is an up-to-date copy of the page on drum storage. Where the hardware installation does not have drum storage this latter bit is always zero. The pointer field contains a null value if the page only resides on disc, a pointer to a drum table entry if the on-drum bit is set or a pointer to a store table entry otherwise. Since drum table entries correspond one-for-one with page-frames on drum the pointer also denotes the position of the page on drum. A drum table entry contains a pointer to a store table entry if there is also a copy of the page in store or a null value otherwise. This is illustrated in figure 3.

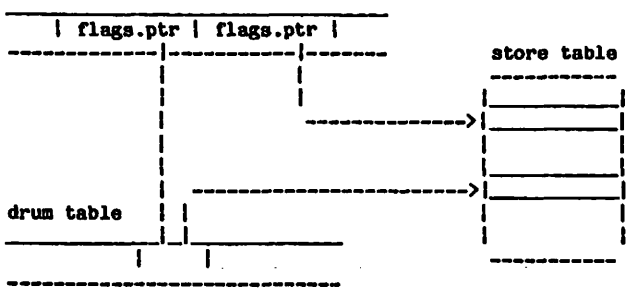

figure 3

Owing to the non-contiguous nature of main store addressing on the 2900 series (each Store Multiple Access Controller (SMAC) has its own range of addresses whether or not they are fully utilised) the position in main store is given in terms of a store table index. Amongst other information, therefore, the store table entry contains the physical

address of the page. In addition to this, each entry contains a count of processes currently sharing the page, a flag field and three link fields. The flags indicate whether a backing store transfer is in progress to or from this page, whether the page has been modified since being paged in (and which must therefore be written out in due course rather than discarded) and whether the page is "recapturable". A page is said to recapturable when it is no longer in use by any process and the page-frame is back on the free page-frame list but has not yet been claimed for use by anyone else. In these circumstances, a page-in request for the page will recapture the page-frame from the middle of the free list and avoid the need for a transfer in of the page from backing store. This mechanism has been observed to have a very significant effect on the performance of the system with figures of typically 30% of all page-in requests being satisfied by recapture even under heavily loaded circumstances and a very much higher percentage under light-load.

The free page-frame list is formed from the store table by linking together those entries not in use. To enable page-frames to be recaptured from the middle of the free list it is constructed with both forward and backward links which can be adjusted appropriately when an entry needs to be removed. The third link in each entry is used in two different ways. When the page is recapturable, it is used to refer back to the AMT entry so that page-frame numbers can be removed therefrom when the page-frame is claimed for a different purpose. When the page is in use and a page-in transfer has been initiated, it is used to point to a list of processes waiting for the page to arrive in store. These are referred to as the page-in-transit lists and for efficiency take the form of messages that will be forwarded via the message-passing mechanism of the kernel to the processes when the transfer in has been completed.

The sequence of operations which take place when a page-in request is made can be summarised as follows. If a page-frame is already allocated for this file page, it may either already be in use by another process or may still be recapturable. If it is recapturable the page-frame is retrieved from the free list and a reply is sent immediately to the requesting process. If it is not recapturable, the flags are examined. If they indicate that a transfer of this file page into store is already in progress, a further pushdown onto the page-in-transit list for that page is made. Otherwise, a reply can be generated immediately even if the page is in transit out of store. In this case the page will not be discarded and the page-frame will not be returned to the free list when the transfer out is completed as the users count will no longer be zero.

If a page-frame has not been allocated to the file page requested, a request is made for a free one. Although, in general, the scheduling manager limits the number of processes in the multiprogramming set so that they may each acquire sufficient real store to accommodate their working sets, occasional delays in being allocated a page-frame still occur. One source of delay arises because processes are given a notional allocation of pages before they are necessarily physically available. For instance, when a process is being paged out of store it's notional allocation is given up and can be reallocated immediately before all the page-out transfers are complete. Another cause of possible delay is when store "overallocation" has been too optimistic. This is described in more detail later. When a page-frame has been allocated, the AMT entry is examined. If the page is marked as new the page is cleared and a reply sent as described above. The "modified" marker in the store table is also set to ensure that the cleared page is

written out subsequently. If the page is not new, the backing store position of the page is determined. If an up-to-date copy of the page exists on drum i.e. the AMT entry is flagged as meaning a drum table entry, a transfer is requested from there. Otherwise, a disc transfer is requested. Since the disc and drum handlers may reorder transfer requests to suit the current status of the device e.g. head position, each request contains an identifier which is returned with the "transfer complete" reply from the device handler. In the case of page-in transfers, the replies are normally sent direct to the originator of the request rather than indirectly through the paging manager. This results in a worthwhile reduction in overhead. Only when a drum read fails for some reason is a reply sent back to the paging manager. In this case, the transfer is requested from the disc site instead and the drum site is marked as "bad". In the case of disc read failure, the originator of the request is informed and an error is signalled to the process. The page may either have been left as it was transferred e.g. a hardware parity error, or cleared to zero for more serious failures.

The philosophy adopted for page-out requests is that the local controller making the request can assume that the paging manager will be able to perform the task without its further involvement. In particular, no reply is made from the paging manager. This simplifies the local controller very considerably and allows the local controller, for instance, to return all its notional resources to the scheduling manager as soon as all the page-out requests have been made following a decision to page its process out from store. This has the effect of maximising the overlap between a process being paged-out and a new process being paged-in since the local conroller of the process being paged-in can make page-in requests immediately. The only slight disadvantage is that very long queues of backing store transfers can build up.

If a page has more than one user, the page will remain in store and no transfers will be requested. The information on whether the page was modified passed with the page-out request is recorded for later use since the remaining process may not modify the page but it will still need to be written out when the process has finished with it if the first process had modified it. If the page only had one user, the page can immediately be paged out. Here another major simplification from EMAS can be observed. This concerns the use of the drum. A decision was made to regard the drum as a higher-speed cache of active disc pages. In particular, when a page is transferred out, it is transferred to both drum and disc, not just to the drum. Subsequent paging-in will normally take place from the drum site, except in the rare case of drum failure, but the very great advantage is that when a section is deactivated or a file is disconnected, the only actions required are those of housekeeping the AMT entries. No transfers have to be made from drum to store and then from store to disc. Furthermore, in the case of a system crash, the disc file site will be more up to date than if many of the modified pages are only on drum which might not be easily recoverable. Clearly this will involve more supervisor overhead in setting up both transfers but this has to be balanced against the avoidance of setting up transfers back from drum to disc and the greater simplicity within the supervisor which will be highly valued during the course of maintenance over the lifetime of the system.

Pages which were not modified since being transferred in will not be transferred out again with the exception that if a valid copy of the page does not yet exist on drum, a transfer out to drum alone is made. When all transfers out have been completed the page-frame is returned to

the free list but marked as recapturable. The "page modified" information is passed with the page-out request from local controller since the "written" markers get set in page table entries and each local controller maintains its own local page tables. Page tables are not used in common when segments are shared. The extra complication of using common page tables is hardly warranted and by not so doing local working set calculations can be made from usage bits (also set in the page tables) without risk of inaccuracy due to sharing. Extra storage will be taken but not by a significant amount.

The system is naturally a good deal simpler when there are no drums but a proportionate increase in main store size is needed to compensate for their absence and maintain the same level of overall performance. Extensive monitoring of the performance of the system was carried out during its development and since it has entered service both by internal measurement and by external measurement using the Edinburgh Remote Terminal Emulator (ERTE) [AD78]. This proved to be of immense value in determining the best approach to certain aspects of the design and demonstrated the most fruitful directions to pursue.

Scheduling Control

One of the fundamental scheduling problems in time-sharing paged systems is that of thrashing, the phenomenon of continual page-faulting from processes that are unable to acquire an adequate number of pages in store. In EMAS 2900, as in EMAS, this is avoided by carefully controlling the number of processes in the multiprogramming set i.e. those processes allocated main store and potentially able to run on the CPU. (The period during which a process is in the multiprogramming set is referred to as a "store residence".) By making an estimate of the number of pages each process requires to run efficiently, the scheduler can ensure that the processes it decides to allow into the multiprogramming set do not overload the available store. The estimate of how much store a process will require the next time it enters the multiprogramming set is made from its previous behaviour. This is a surprisingly good indicator but clearly a process will occasionally change context and begin to exhibit different characteristics. For this reason the scheduler is designed to be adaptive and changes of behaviour are dealt with automatically. No intervention from either the user himself, his process or the system manager is required. The adaptation is derived from a table of categories through which processes migrate. Each category defines a combination of store requirement and CPU-time requirement together with a priority, a set of transition indicators and various other data. The system attempts to be democratic in the sense that categories which define lower resource requirements are allocated higher priorities and vice versa. The transition indicators control the route through the categories a process takes depending on the variations in its resource requirements as time goes by. A typical category table is shown in figure 4.

The category table shown is a fairly basic one that has undergone no tuning to fit a particular installation and the known characteristics of the programs which run on it such as popular editors and compilers. By being table-driven, the scheduling is very easy to change. Deciding what changes are sensible is a more difficult problem. The system-wide ramifications of local scheduling changes are not always immediately apparent. The regular use of the ERTE remote terminal emulator system was of particular value in this respect. One modification which is

category	pages	time	priority	more pages	more time	less pages	runq 1	runq 2	strobe
1	20	4	1	2	5	1	1	1	0
2	32	4	2	3	6	1	1	1	0
3	64	4	2	4	7	2	1	1	0
4	128	4	3	4	8	3	1	1	0
5	20	24	2	6	9	5	1	2	0
6	32	24	2	7	10	5	1	2	0
7	64	24	3	8	11	6	1	2	0
8	128	24	4	8	12	7	1	2	4
9	20	64	3	10	9	9	2	2	0
10	32	64	4	11	10	9	2	2	8
11	64	64	4	12	11	10	2	2	8
12	128	64	5	12	12	11	2	2	8

figure 4

normally present is to incorporate into the table a disjoint set of categories and their transitions suitable for jobs submitted to a batch stream. EMAS 2900 has facilities which allow a number of processes to be created as batch processing streams but these will need different priorities so as not to interfere with the response of interactive processes. When a process is placed in a certain category the scheduler tells the local controller the number of pages it may acquire on behalf of the process and the number of time-slices of CPU it may consume (the "pages" and "time" columns of the table). The local controller can then allow the process to proceed within these limits. If these resources are not sufficient, the local controller is obliged to request to be rescheduled. In principle this means that that process will be removed from the multiprogramming set in order to give other processes a chance to make progress. In asking to be rescheduled, the local controller says which resource it has run out of and how much of the other resource it had used by this stage. This allows the scheduler to adapt the category of the process to one which will be more suitable. For example, if the process had run out of time, the scheduler would assign it to a new category that had a greater allocation of CPU-time. This category is indicated in the "moretime" column of the category table. In addition, if there is a category with fewer pages but still with more pages than the process had used when it ran out of time, that category would be chosen instead in order to minimise overall resource allocation and to maximise the priority of the process.

Local controllers endeavour to avoid exceeding their page allocations to the extent of performing "strobe" operations through the page usage markers in their page tables at certain times. If the marker indicates that a page has not been accessed recently then it can be paged out. The typical time this is done is when the page allocation has already been used up and a further page-fault has occurred. This is only attempted once to avoid local thrashing occurring. Strobing also occurs at other times controlled by the category in which the process resides. This control is necessary because the effectiveness of strobing varies considerably depending on the characteristics of the process and because strobing is a relatively time-consuming operation to perform.

The priority given to a process from its category is used by the scheduler exclusively to control when it is admitted to the multiprogramming set. The scheme is not pre-emptive but uses a

"priority ratio" table which indicates the priority of process that the scheduler should load next. Even if the process at the head of the corresponding priority queue cannot yet be loaded, no other smaller process is loaded in its stead. This table contains a sequence of priorities with the higher priorities occurring more often and lower priorities occurring less often. Thus higher priority processes will be admitted frequently but the lower priority processes can never be entirely excluded from the chance of making forward progress in their computation. A process is allowed to change category "on-the-fly" and remain in the multiprogramming set when no other process is waiting on a higher priority queue. This tends to occur when the system is lightly loaded and avoids the cost of paging a process out and in again unnecessarily.

There are two stages to admitting a process to the multiprogramming set. The first is to allocate pages for and to page in the local controller stack (the rest of the local controller is shared and already resident). The second is to allocate the pages for the process itself and activate the local controller which then passes control to the process and deals with its page-faults. Since the local controller stack is only three pages long, an allocation for this can almost always be made well ahead of sufficient pages becoming available for the remainder of the allocation. The scheduler then immediately pages in the local controller stack using exactly the same mechanisms and facilities described above that local controllers themselves use. The pages are thus very likely to be store by the time the full allocation is available and the potential inefficiency of two stages is largely avoided.

In EMAS and in the initial version of EMAS 2900, a "working set reloading" scheme (also known as "preloading") was used (see [AD77]). In other words, the constitution of the working set of each process was remembered at the end of a store residence and these pages were automatically paged in again at the start of the next store residence without waiting for page-faults. On the 4-75s this scheme was very beneficial in improving response. On the 2900s it proved to be rather less so for various reasons. Firstly, the author was tempted to implement a rather complex arrangement involving three stages of loading a process into store, the extra stage being the allocation of just that part of the total allocation of pages required for the local controller to reload the working set, followed by a pause until the remainder became available before the process was allowed to run. It was also discovered that on machines with drum storage at least there was very little performance gain since the drums were so fast that a page fault could be satisfied very quickly. The inaccuracy of preloading i.e. those pages reloaded but never actually accessed, was for some unexplained reason also rather higher on the 2900s than it had been on the 4-75s which resulted in more wasted effort. Preloading was therefore removed and a rather simpler multi-stage store allocation scheme implemented instead. Having dealt with the local controller allocation, the scheduler attempts to allocate the whole amount required in the category. If there is not sufficient to do this, then what there is is allocated anyway and the local controller set running with the knowledge that it has not yet received its whole allocation. When and if it runs out of this partial allocation it can request the remainder from the scheduler. The scheduler may by this time have made a further allocation to the process in which case the local controller can carry on again. If not, the local controller is held up until more becomes available. By this means, all the spare allocatable store is mopped up and at least the satisfaction of the initial page faults of the process

is set in motion as soon as possible. In the absence of drum storage the benefits of preloading might have been expected to reassert themselves but the substitution of large main stores meant that page recapture tended to be much more effective and to perform pretty much the same function.

The method of totting up the process page allocations so as not to overload the available store is effective but has drawbacks. One is that most if not all processes will share pages in store from common files. Indeed, much of the effectiveness of the system derives from this. Clearly, only one actual page will be needed for each such shared page but as many notional pages as processes sharing it will have been allocated. If actual store pages are available corresponding with all the notionally allocated pages then store pages will be unused and wasted. This can easily be overcome by adding one to the number of pages available to be notionally allocated each time a page sharing occurs and subtracting one when a shared page is released. A second drawback arises from the fact that in order for a process to adapt itself quickly to a reasonable category, the page allocations in categories rise in fairly large jumps. Any particular process is liable, therefore, not to be using the whole of its allocation at any point in time i.e. a form of internal fragmentation. This disadvantage can be mitigated by overallocating the notional allocations to a limited extent, say twenty per cent. Since requests for store pages cannot always be immediately satisfied (a previous process may still be being paged out as mentioned above) a queueing mechanism for requests already exists. With overallocation more queueing takes place but the overall gain is noticeable. A further snag is unfortunately introduced though. This is the possibility of deadlock arising. It may occasionally be the case that the processes in the multiprogramming set are so near to their notional page allocation limits that no store page-frames remain to be allocated. If no more page-frames are expected to be released when page-outs complete and all the processes are requesting a further page (via the paging controller) then a deadlock situation has arisen. As long as the overallocation is kept reasonably small such deadlocks only occur rarely. When one does occur the solution we adopted is to detect that one has occurred in the store page allocation routine and arbitrarily to force one local controller to page out its process. This will release pages for the remaining processes to proceed. This is admittedly somewhat messy but effective. A final imperfection of totting up is that unless some extra action is taken the store may become dominated by large processes even though they have low priority. This would have they effect of limiting the number of small interactive processes in the multiprogramming set and correspondingly limit their overall rate of response. This can be overcome by arbitrarily limiting low priority processes to a number which do not occupy more than some proportion such as a half of the total store.

When a process wishes to wait for some external event such a console input, the local controller has to request the scheduler to suspend it. If there is sufficient store available the scheduler may allow the process to "snooze" in store without being paged out but otherwise the local controller will have to page its process out before suspending. An intermediate form of snoozing has also been implemented which consists of just retaining the local controller stack. This occupies very much less space and still improves response to the user when the process is unsuspended. When suspended, a message to a process causes the scheduler to be invoked so that the process can be rescheduled and brought back into the multiprogramming set in due course to receive the message. The message passing scheme EMAS 2900 adopted is very similar

to that described in [WH73] for EMAS but a certain amount of extra flexibility has been built in for paged processes. Basically, messages are addressed to a particular service number but any one service routine may accept several service numbers. Service numbers can also be "inhibited". This inhibits the message dispatcher from delivering messages on that number. A typical usage of this facility is when a device handler cannot deal with any more transfer requests for the moment. Paged processes each have four service numbers, one for the local controller to receive messages on and three for the paged part of the process. One of these is reserved for asynchronous events and the other two for all other events and messages. These latter two allow the process to control what messages get delivered to it. A common inconvenience in operating system implementation is for a process to be obliged to service any type of message which is sent to it at any time. A basic loop of some description is often used which introduces problems of modularity. In EMAS 2900, by inhibiting one of the pair of service numbers, messages to that number can be held pending while messages sent to the other can still be received. By controlling what service numbers other processes pass their messages on, message handling code can be modularised much more conveniently with a corresponding gain in clarity and simplicity.

Asynchronous messages when passed to a process result in a different context being set up from a stack of contexts set up for the purpose by the process. We have not been entirely happy with this scheme although some mechanism of the sort is clearly required. In any future implementation, we would favour a user having much more freedom to use multiple concurrent processes. This then leads naturally to the abandonment of asynchronous events as such in that it should be possible to arrange that all events are synchronous to some sub-process or other. A system such as that would be much cleaner and less error-prone than our present arrangement.

Dispatcher priorities amongst processes in the multiprogramming set use the "runq1" and "runq2" fields from the category table. The dispatcher maintains two run queues for paged processes (they are, in fact, message queues and the message dispatcher is one and the same thing as the paged process dispatcher). Processes on the first have pre-emptive priority over those on the second and which queue processes go on is defined by "runq1" and "runq2". "runq1" indicates the queue that the process is to go on during its first time-slice in the round-robin CPU-allocation scheme used for the processes in the multiprogramming set. "runq2" indicates the queue it should go on during the second and subsequent time-slices of its residence. This control allows highly interactive processes to be given favoured treatment and improves their response times significantly. Processes which only ever go on the second queue still make reasonable forward progress since the processes on the first queue are such that they do not hog the CPU. Most of them usually do not even use up the complete time-slice.

The Local Controller

The main functions of the local controller have in essence been described above in the contexts of paging and scheduling control. The organisation of its private data structures and other details remain to be described.

As has also been described above, there is an incarnation of a local controller for each virtual process. This incarnation consists of shared code which is common to all local controllers and a private data segment suitably protected from user access by using the "rings of protection" mechanism provided by the 2900 series hardware. This segment is used to hold both the local segment table for the process and the stack which is associated with running the local controller as an IMP program.

Since a local controller is intimately associated with the global controller it was found convenient to compile the local controller code as a procedure within the kernel rather than as a separate entity. This implies that whenever the local controller procedure is called the stack has to be switched from the one on which the kernel is running to the local controller's own stack. Doing this was not as straightforward as one might imagine since control of stacks is highly protected by hardware mechanisms which ensure the integrity of a running system. The scheme which was evolved results in the local controller only being called as a procedure on the first occasion i.e. on process creation. Thereafter, the process "activate" instruction is used to re-enter the local controller at the point where it exited previously. The great advantage of this scheme is that all the data structures required by the local controller can be allocated by the normal IMP mechanisms and on a stack which can be paged out when the process is not in the multiprogramming set. The kernel stack still remains resident. A subsidiary advantage is that the local controller can access global controller data structures just as global variables, even though they are on a different stack. Philosophically this is perhaps undesirable but makes life very much simpler in one or two instances. By being able to page out much of the information which the kernel has to maintain about each process i.e. all the local controller stack data, the amount of resident storage which is required for each active process is reduced to a minumum. In our case case this is just a few words of storage.

In order to access a file it is mapped into virtual space by being "connected". (This is one of the functions of the file system.) To make a connection the director writes mapping information into the local controller's data structures. A segment is made accessible to the director for this purpose and the local controller tells the director where the data structures are within this segment on start-up. This requires cooperation between the two layers of the system, local controller and director, to avoid inadvertant misuse but has not proved to be a problem from the reliability point of view. The data structures which define connection mappings consist of a table known as the "secondary segment table" parallel to the local segment table, and a series of lists each cell of which describes a section of storage on disc. Figure 5 shows the organisation.

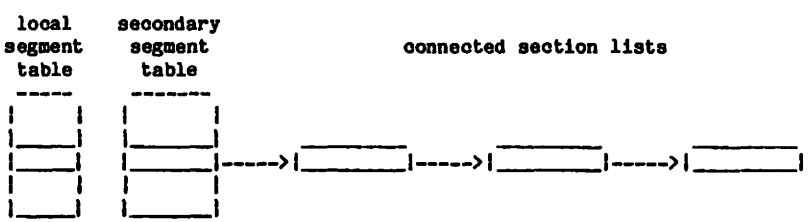

figure 5

The description of each section of storage contains the disc address of its start, its length and a field which is either used to hold the "new" bits for pages in the section when it is not active or the "amtx" active memory table index received from the global controller when it is active. The remaining tables maintained by the local controller are concerned with active sections. For each active section, those pages within it which are also active i.e. in use by this process, must be remembered. Furthermore, an abbreviated history of the usage of active sections must also be maintained. The structure of the tables went through several iterations before settling on those now to be described. These iterations were entirely local to the local controller and therefore posed no interface problems with the global controller. This was another benefit of the modularisation into local and global controllers. Originally, linked lists were extensively used but these proved to take more space than was desirable and so bit maps were substituted which were more compact. Fortunately, the 2900 series order code contains some rather useful instructions for efficient bit manipulation and these were made use of in the local controller. IMP has a convenient facility for embedding machine instructions in-line with the high-level code which allows normal IMP variable and label names to be incorporated in the instructions. The system implementors' philosophy since the implementation of EMAS has been to program initially completely in IMP and only to embed machine instructions where absolutely necessary for reasons of efficiency or for input-output initiation etc.

The main bit map consists of an array of 64-bit words, declared as a %longintegerarray in IMP, each word of which relates to some segment and each bit to a page within a segment. It is essentially a "tertiary segment table" therefore but not in parallel with the primary and secondary tables. One of these words is set aside when any section within a segment is first activated. The local controller then regards these as active segments. The word to allocate is chosen arbitrarily using another word as a bit map to indicate which are in use. Entries in the array are indexed from a spare field in the local segment table as shown in figure 6.

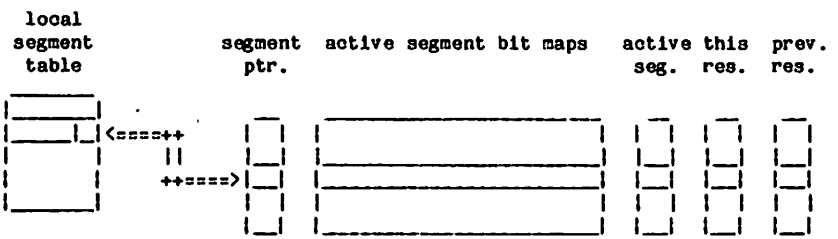

figure 6

The segment pointer array shown relates active segment bit map words back to segment numbers. The rightmost parts of the data structure shown are updated to indicate those segments which contain pages that have been referenced during the present store residence and those segments pages of which were only referenced in a previous residence. These latter are used in the management of local active store by the local controller. The active segment bit map array was chosen to be 32 entries long. This allows up to 32 segments to be active at any one time, a limit which has proved more than adequate in practice.

Nevertheless, as the context of a process changes this limit would be exceeded through old segments still being active although not in current use. The local controller therefore attempts to work out a form of working set of active segments and to deactivate sections within segments that are no longer within that set. Apart from wishing to economise on bit map space in the local controller stack, the amount of space needed by the global paging manager in its active storage tables is also directly related to the total number of active segments in all the local controllers.

The construction of page tables for each of the segments of local virtual space in use is a further function of the local controller. For this purpose the local controller is allowed to claim physical page-frames directly from the global allocator instead of indirectly through the paging manager but they still must be accounted for in the local controllers notional page allocation. Several maximum sized page tables can be packed into one page and many more when the segments are of the typical small size.

The remaining main function of the local controller is to handle communications with the director of the process. Interprocess messages are forwarded by the local controller, for instance, together with an inevitably growing list of specialised services that have been thought to be desirable as the system has developed.

Conclusions

The exercise of transporting EMAS to the 2900 series hardware has been a conspicuous success as far as its users are concerned. It has proved much more effective than the manufacturers operating system in a University environment mainly, we believe, because our objectives were clear and because we did not aim to produce a system that was all things to all men. The opportunity to rethink the design of the kernel after a number of years experience with EMAS was invaluable and the resulting design has improved on the original in almost every respect. The simplicity and elegance that we sought has largely been achieved. Though one can always think of ways one could do it even better next time by and large we are satisfied with the outcome of our efforts. It is hoped that a future paper will update the description of the kernel given here to include recently added features such as those dealing with multi-processor configurations.

Acknowledgements

The implementation of a large system such as EMAS 2900 is very much a team effort. The constitution of our team changed over the period of the development but the hard core of the implementors who stuck to the task deserve special mention for their efforts and constant availability to discuss ideas and develop the EMAS philosophy. They were P.D.Stephens, J.K.Yarwood and N.H.Shelness together with the able assistance of W.A.Laing, R.R.McLeod and F.Stacey.

References

[AD77] J.C.Adams:"Performance measurement and evaluation of time-shared virtual memory systems", Ph.D. Thesis, Edinburgh University, 1977.

[AD78] J.C.Adams, W.S.Currie and B.A.C.Gilmore: "The structure and uses of the Edinburgh Remote Terminal Emulator", Software Practice and Experience, Vol.8, 451-459 (1978).

[BU78] J.K.Buckle: "The ICL 2900 Series", Macmillan, 1978.

[DE78] P.J.Denning: "Working Sets Past and Present", Purdue University Report CSD-TR-276, 1978.

[LA81] W.A.Laing: "Communications Control in the Operating System EMAS 2900", presented at IUCC, September 1981.

[RE75] D.J.Rees: "The EMAS Director", Computer Journal, Vol.18, 122-130 (1975).

[ST74] P.D.Stephens: "The IMP language and compiler", Computer Journal, Vol.18, 131-134 (1975).

[ST80] P.D.Stephens, J.K.Yarwood, D.J.Rees and N.H.Shelness: "The evolution of the Operating System EMAS 2900", Software Practice and Experience, Vol.10, 993-1008 (1980).

[WH73] H.Whitfield and A.S.Wight: "EMAS - the Edinburgh Multi-Access System", Computer Journal, Vol.18, 331-346 (1973).